BrainGoated - Project Final Report

CMPT 276 - D200: Introduction to Software Engineering

Manjari Prasad

Alisha Mahebub Jesani

Gurpreet Kaur

Beyzanur Kuyuk

GitHub Repository : final-project-11-stars

https://github.com/CMPT-276-SPRING-2025/final-project-11-stars.git

https://braingoated.netlify.app/

https://youtu.be/hO3A0SJO3FQ

Analysis of the Project's Success in Meeting User Needs

User Needs and Fulfilment:

The BrainGoated application was developed to address the growing demand for educational platforms that are both engaging and accessible, especially for children aged 8–16. Traditional learning tools often lack interactivity and customization, which can reduce student motivation. BrainGoated aims to provide a solution by offering dynamic, personalized quizzes powered by external APIs and enriched with educational feedback mechanisms.

As outlined in the project proposal, the platform targeted three core user groups (students, parents, and teachers), each with distinct needs:

1. Students sought fun, interactive quizzes tailored to their interests.
2. Parents valued a safe, easy-to-use platform that supports learning without requiring extensive setup.
3. Teachers required tools that could complement class material and engage students actively.

These needs were met through the following key features:

1. Text- and image-based quizzes tailored by category and difficulty.
2. The ability to create your own quiz in the category, difficulty level, and language of the users' choice.
3. AI-generated fun facts and Bud-E chatbot explanations to encourage deeper understanding.
4. Language selection options, enabling quizzes to be played in several supported languages.
5. Dark mode and mobile responsiveness to enhance accessibility and usability across devices.

User Feedback and Testing Session:

Following peer testing with classmates, feedback was collected through a structured Google Form. The responses highlighted several key strengths:

1. The majority of users rated the design as "Excellent".
2. Navigation was considered "Very Easy" or "Somewhat Easy" by most participants.
3. The platform was described as “Very Engaging”, with positive remarks on the theme, color palette, and interactivity.
4. The dark mode feature was widely appreciated for visual comfort.

Users found the content appropriate and fun, with several noting they would recommend the platform to others.

In addition to classmate feedback, the application was tested with actual children in the target age range. These sessions provided valuable insights and validation. The children found the platform intuitive, visually exciting, and highly enjoyable. They particularly liked the ability to choose quiz topics, and the Bud-E chatbot, which made the learning process feel more like a game than a test.

This hands-on testing confirmed that the platform resonated with its intended audience, not just in theory but in actual use.

Areas of improvement identified during testing, such as the help icon not functioning consistently and the homepage scroll not being obvious, were promptly addressed. Animations and bounce indicators were added, and strong interactivity was ensured.

The BrainGoated platform has demonstrated strong alignment with user expectations. By responding to real feedback, refining UI/UX components, and maintaining a kid-friendly tone, the project has successfully delivered a solution that is both educational and enjoyable.

Software Development Lifecycle (SDLC) Model

For the BrainGoated project, the team adopted an Agile-**Kanban** software development life cycle (SDLC) model. This approach was chosen for its flexibility, visual task tracking, and its support for continuous, incremental development, which was essential for responding effectively to both peer and user feedback.

Implementation of the Agile–Kanban Model:

The Kanban board was hosted on GitHub. The board included custom columns for:

1. To Do: Initial backlog and planned tasks
2. In Progress: Active development tasks
3. Completed: Finalized tasks, ready for deployment

This structure allowed the team to monitor progress in real time, assign responsibilities, and shift priorities as needed.

SDLC Phases and Execution:

1. *Planning*:
    a. Defined user personas and system requirements as per our project proposal report
    b. Selected primary APIs (The Trivia API and OpenAI API) and outlined their integration.
    c. Created a detailed Work Breakdown Structure (WBS) and development timeline.

2. *Design*:
    a. Low- and high-fidelity prototypes were created in Figma to visualize user flows and UI components.
    b. UI mockups included interactive elements like quiz cards, pop ups, and the Bud-E chatbot interface.

3. *Development*:
    a. The application was built using React.js for structure and logic, with CSS for styling.
    b. Features were built incrementally, tracked via the GitHub Issues and linked to Kanban cards.
    c. Each feature (e.g., dark mode, mobile responsiveness, create your own quiz, score tracker) was implemented as a separate task to maintain modularity and clarity.

4. *Testing*:
    a. A component-focused, behavior-driven testing approach, emphasizing real user behavior, using Jest and React Testing Library.

    b. Continuous testing was carried out for each component via manual QA and classmate peer reviews.
    c. Issues identified (e.g., scrolling not obvious, chatbot visibility) were logged and resolved through Kanban updates.

5. *Deployment*:
    a. The project was deployed using Netlify, with version control and CI handled via GitHub.
    b. Documentation was continuously updated in the GitHub repository and final milestone deployments were aligned with internal deadlines.

These iterative changes were documented and updated directly within the GitHub Kanban board, reinforcing the effectiveness of Agile-Kanban as a flexible SDLC model for this project.

The Kanban-based Agile model proved highly effective for BrainGoated. It enabled continuous tracking of feature development, rapid integration of peer and user feedback, and dynamic prioritization of tasks. The development process remained user-focused, collaborative, and adaptive, resulting in a final product that was polished, robust, and closely aligned with the project's original goals.

Features Implemented for Each API

The final version of BrainGoated integrates two core APIs: *The Trivia API* and *OpenAI API*, each contributing distinct functionalities to the application.

<u>The Trivia API:</u>

1. Category and Difficulty Selection: Enables users to choose quiz topics and difficulty levels to personalize their experience.
2. Text-Based and Image-Based Questions: Supports both question formats to cater to different learning styles.
3. Language Support: Questions can be fetched in multiple languages including English, French, Hindi, German, and more.
4. Randomization: The API ensures varied content across quiz sessions.

Feature Changes and Additions:

1. The “quiz creation” feature originally planned for teachers was removed due to Trivia API limitations.
2. Language selection was introduced based on accessibility goals.

OpenAI API:

1. AI-Generated Quizzes: Allows users to create quizzes on niche topics, preferred difficulty levels and language.
2. Fun Facts Generator: After each question, the system fetches a fun fact related to the topic to enhance educational value.
3. Bud-E Chatbot: Users can ask follow-up questions and receive detailed yet simplified AI explanations.

Feature Changes and Additions:

1. Language customization is also available with the custom quiz generation
2. The "Learn First" feature was consolidated into dynamic fun facts to avoid disrupting the quiz flow.
3. Bud-E was expanded to appear as a floating popup with animations and a tooltip, improving its discoverability.

Continuous Integration, Deployment, and Monitoring

CI/CD Pipeline and Monitoring (Overview):

The CI/CD pipeline for BrainGoated was implemented using GitHub Actions for automation, with Jest integrated as the testing framework to validate code quality during the pipeline. The project is deployed to Netlify, and UptimeRobot is used to monitor application availability. This setup maintains a high standard of code quality, reliability, and performance throughout the development lifecycle by automating the processes of building, testing, and deploying the project.

The pipeline is defined in a `.github/workflows/ci-cd.yml` file within the repository. This YAML configuration specifies the jobs, triggers, and environment setup used during continuous integration and deployment.

Continuous Integration (CI):

```yaml
name: CI/CD Pipeline for BrainGoated

on:
 push:
   branches:
     - main
 pull_request:
   branches:
     - main
```

Continuous integration is facilitated through GitHub Actions, which automatically triggers workflows whenever:

- Code is **pushed** to the main branch
- A **pull request** is made to the main branch

This ensures all new changes go through automated checks before they are deployed.

Build and Test Phase:

```yaml
jobs:
 build-and-test:
   runs-on: ubuntu-latest

   steps:
     - name: Checkout Repository
       uses: actions/checkout@v3

     - name: Set up Node.js
       uses: actions/setup-node@v4
       with:
         node-version: 20

     - name: Cache node modules
       uses: actions/cache@v3
       with:
         path: ~/.npm
         key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
         restore-keys: |
```

```
        ${{ runner.os }}-node-

    - name: Install Dependencies
      run: npm install

    - name: Run Tests
      run: npm test -- --watchAll=false

    - name: Build Project
      run: npm run build
```

The first part of the workflow, defined in the .yml file under the build-and-test job, is responsible for checking if the code runs properly and passes all tests. As mentioned above, this phase is only triggered when changes are pushed or a pull request is made to the main branch. Here's how it works:

1. *Environment Setup:*

   ```
   jobs:
    build-and-test:
      runs-on: ubuntu-latest
   ```

   The pipeline starts by setting up a clean virtual machine using `ubuntu-latest` as the operating system.

2. *Checkout Repository (uses: actions/checkout@v3):*

   ```
      steps:
        - name: Checkout Repository
          uses: actions/checkout@v3
   ```

   It pulls the latest version of the project from GitHub into the runner environment so the workflow can access the source code files for testing and building.

3. *Set up Node.js:*

```
    - name: Set up Node.js
      uses: actions/setup-node@v4
      with:
        node-version: 20
```

Sets the Node.js version to 20 using the `setup-node` action, ensuring the workflow uses the same runtime environment as the project during builds and tests.

4. *Cache Node Modules (uses: actions/cache@v3):*

```
- name: Cache node modules
      uses: actions/cache@v3
      with:
        path: ~/.npm
        key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
        restore-keys: |
          ${{ runner.os }}-node-
```

To speed things up, it caches the `node_modules` folder based on the `package-lock.json` file using `actions/cache`. This helps avoid reinstalling the same packages every time. When the dependencies haven't changed (package-lock.json was not updated), the workflow skips redownloading them and instead reuses the cached versions, making installs faster and more efficient.

`path: ~/.npm` :

Specifies the folder to cache - in this case, the local npm cache directory

`key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}`:

Generates a unique key for the cache using the operating system and a hash of the `package-lock.json` file. If the dependencies change, a new key is generated, and the cache is refreshed.

```
restore-keys: |
      ${{ runner.os }}-node-
```

Provides a fallback option. If the exact key isn't found, GitHub tries to match using just the OS to reuse older caches if possible.

5. *Install Dependencies:*

```
    - name: Install Dependencies
      run: npm install
```

Installs all necessary packages using `npm install`, ensuring they are available before testing and building.

6. *Run Tests (run: npm test -- --watchAll=false):*

```
- name: Run Tests
      run: npm test -- --watchAll=false
```

Executes the test suite using `npm test`, allowing verification of the codebase. The `--watchAll=false` flag ensures that Jest runs the tests once and exits rather than entering watch mode — which is intended for local development and would otherwise cause the pipeline to hang/stall.

The deployment phase is configured to depend on the success of the testing job, ensuring that deployments are only made when all tests pass. This prevents broken or unverified code from reaching the production environment.

7. *Build Project:*

```
    - name: Build Project
      run: npm run build
```

After all tests pass, the application is compiled into a production-ready version using `npm run build`. This step generates static files inside the `/build`

directory. While these files are not transferred directly to the deployment stage (which performs its own build), this step serves as a confirmation that the build process completes successfully in a clean environment and can catch any build-time errors early in the pipeline.

Continuous Deployment (CD):

The deployment phase is configured to depend on the entire `build-and-test` job. If either the tests fail **or** the build fails, the deployment phase will be skipped.

Deployment Job:

```
deploy:
   needs: build-and-test
   runs-on: ubuntu-latest
   if: github.ref == 'refs/heads/main'

   steps:
     - name: Checkout Repository
       uses: actions/checkout@v3

     - name: Set up Node.js
       uses: actions/setup-node@v4
       with:
         node-version: 20

     - name: Cache node modules
       uses: actions/cache@v3
       with:
         path: ~/.npm
         key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
         restore-keys: |
           ${{ runner.os }}-node-

     - name: Install Dependencies
       run: npm install

     - name: Build Project
       run: npm run build

     - name: Install Netlify CLI
       run: npm install -g netlify-cli

     - name: Deploy to Netlify
       env:
         NETLIFY_AUTH_TOKEN: ${{ secrets.NETLIFY_AUTH_TOKEN }}
         NETLIFY_SITE_ID: ${{ secrets.NETLIFY_SITE_ID }}
       run: netlify deploy --prod --dir=build --auth=$NETLIFY_AUTH_TOKEN
--site=$NETLIFY_SITE_ID
```

1. *deploy*:

```
deploy:
   needs: build-and-test
   runs-on: ubuntu-latest
   if: github.ref == 'refs/heads/main'
```

Declares the name of the deployment job. This section starts a new job in the pipeline responsible for deployment.

```
needs: build-and-test
```

Indicates that this job is dependent on the successful completion of the `build-and-test` job. If tests or builds fail, deployment will not proceed.

```
runs-on: ubuntu-latest
```

Sets the virtual environment where the deployment job will run. GitHub uses a clean Ubuntu-based runner for consistency.

```
if: github.ref == 'refs/heads/main'
```

Ensures that deployment only occurs when changes are pushed to the `main` branch. This prevents accidental deployments from feature or testing branches.

2. *steps*:

```
steps:
     - name: Checkout Repository
       uses: actions/checkout@v3

     - name: Set up Node.js
       uses: actions/setup-node@v4
       with:
         node-version: 20

     - name: Cache node modules
       uses: actions/cache@v3
```

```
      with:
        path: ~/.npm
        key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
        restore-keys: |
          ${{ runner.os }}-node-
```

`name: Checkout Repository`

Pulls the latest code from the repository. Even though the code was checked out earlier in the `build-and-test` job, it must be done again because each job runs in isolation.

`name: Set up Node.js (uses: actions/setup-node@v4)`

Sets the Node.js version to 20 using the `setup-node` action, ensuring the workflow uses the same runtime environment as the project during builds and tests.

`name: Cache Node Modules`

Same steps as defined in the `build-and-test` job: Caches the npm dependency directory to speed up future installs. If the lock file hasn't changed, it reuses cached packages instead of reinstalling.

3. *Install Dependencies (run: npm install):*

```
    - name: Install Dependencies
      run: npm install
```

Installs all necessary packages using `npm install`, ensuring they are available before testing and building.

4. *Build Project:*

```
    - name: Build Project
      run: npm run build
```

The application is built using the `npm run build` command to produce the most recent production-ready files in the `/build` directory, which is used for deployment. This step ensures that the deployment reflects the latest validated code.

5. *Install Netlify CLI:*

```
    - name: Install Netlify CLI
      run: npm install -g netlify-cli
```

This step installs the Netlify Command Line Interface (CLI) globally (`-g` flag) in the CI/CD environment. Installing it globally ensures the CLI is available as a system-wide command, allowing the workflow to run deployment commands like `netlify deploy` later in the pipeline.

   a. The Netlify CLI provides access to Netlify's platform from the command line, which is essential in this context because:
   b. It enables automated deployments from the CI pipeline without manual interaction.

   c. The CLI can authenticate and push build artifacts to the correct site using the `--auth` and `--site` flags.

   d. It integrates seamlessly with secret tokens to ensure secure, headless deployment to production.

6. *Deploys to Netlify:*

```
    - name: Deploy to Netlify
      env:
        NETLIFY_AUTH_TOKEN: ${{ secrets.NETLIFY_AUTH_TOKEN }}
        NETLIFY_SITE_ID: ${{ secrets.NETLIFY_SITE_ID }}
      run: netlify deploy --prod --dir=build --auth=$NETLIFY_AUTH_TOKEN
--site=$NETLIFY_SITE_ID
```

   a. This is the final step in the deployment job and is responsible for pushing the production-ready build of the application to Netlify, making it live.
   b. `env` :Defines environment variables for secure authentication.

c. `${{ secrets.NETLIFY_AUTH_TOKEN }}`: A secret token that authenticates the workflow with our Netlify account. This ensures the deploy action is authorized.
d. `${{ secrets.NETLIFY_SITE_ID }}`: The unique ID of our Netlify site, telling Netlify exactly where to deploy the build.
e. `netlify deploy --prod`: Deploys the site to production rather than a draft or preview version.
f. `--dir=build`: Specifies the output directory (/build) containing the static site files generated by `npm run build`.
g. `--auth=$NETLIFY_AUTH_TOKEN`: Passes the auth token securely.
h. `--site=$NETLIFY_SITE_ID`: Identifies the exact Netlify project to deploy to.

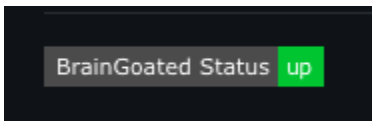
This step ensures that only **tested, built, and validated code** is deployed to the live environment. By using secrets stored in the GitHub repository, it keeps sensitive data secure while still enabling full automation of the deployment process.

Deployed site: https://braingoated.netlify.app

Website Monitoring (UptimeRobot):

To ensure ongoing reliability and to track system health post-deployment, **UptimeRobot** was integrated into the monitoring strategy.

- UptimeRobot sends an HTTP request to https://braingoated.netlify.app every **1 hour,** checking the application URL.
- If the site is ever down, it can notify us through email or alerts (we're using the default free tier)
- A **public status page** is generated and hosted at stats.uptimerobot.com/I4KtZdeNvh, allowing transparent reporting of operational status to users.
- Additionally, a dynamic **status badge** has been embedded in the project's README file. This badge uses the UptimeRobot monitor ID and API key to reflect the real-time operational state of the website, always shows whether the site is "Up" or not:



This setup provides a comprehensive end-to-end CI/CD pipeline that includes continuous testing, automated deployment, and real-time uptime monitoring, ensuring both developer confidence and user trust.

Project Testing Strategy

Our testing strategy for the BrainGoated project was built around ensuring confidence in both UI functionality and state-dependent logic, especially since the app relies heavily on React Context, user interactions, timing-based animations, and dynamic content rendering from APIs.

Testing Philosophy:
We adopted a component-focused, behavior-driven testing approach, emphasizing real user behavior. Our toolset included:

1. Jest for test running, assertions, and mocking
2. React Testing Library for simulating user interactions and querying the DOM
3. Context mocking to isolate component logic from global state
4. Fake timers to simulate animations and timeouts deterministically

This approach allowed us to test features as users would experience them, without being tightly coupled to implementation details.

What is Tested?

HomePage:

1. Rendered key elements like headings, branding, and the animated Bud-E iframe.
2. Mocked `<Quizcategories />` to isolate page layout testing.
3. Verified dark mode toggle updates `localStorage` and applies correct CSS classes.
4. Simulated auto-scroll after 3.5s using `jest.useFakeTimers()` with a mock `scrollIntoView().`

QuizCategories:

1. Rendered all 11 quiz categories and verified accessibility roles.
2. Handled custom vs regular categories using modal logic.
3. Tested form validation errors (e.g., missing difficulty or language).
4. Mocked `QuizContext` to simulate quiz mode switching and state management.

QuizPage:

1. Covered multiple states:
    a. Loading animation
    b. Error fallback view
    c. Question and options display

2. Verified both correct and incorrect answer flows with UI popups, score updates, and Bud-E explanation bubbles.

3. Mocked:
    a. `useNavigate` for routing
    b. `canvas-confetti` for win animations
    c. `getExplanation()` (OpenAI API) and `getBudEReply()` for Bud-E chatbot

4. Used `act()` and `await` for managing popup animation delays and API call resolutions.

ResultPage

1. Verified score rendering, motivational messages, and fallback when no answers are submitted.

2. Implemented and tested:

    a. Filtering by correct/incorrect/all questions
    b. Displaying **text and image answers**, including those parsed from JSON
    c. Displaying explanations
    d. Navigation back to category page and resetting quiz state

Footer

1. Verified rendering of text: team credits, current year copyright.
2. Checked the presence of navigation links (About, Contact).
3. Confirmed Bud-E mascot image is present with alt text.

HelpButton

1. Rendered a toggleable help button with accessible icon.
2. On click, displays a usage guide with list items.
3. Clicking "Got it!" closes the popup cleanly.

API Testing:

We successfully implemented unit tests for both major APIs in the app.

- *Trivia API –* `/utils/triviaApi.js`

  Purpose: Fetch quiz questions dynamically from `https://the-trivia-api.com`.

  Tested:

  1. Correct URL construction with dynamic category/difficulty/language
  2. Proper handling of success (`res.ok`) and failure (`!res.ok`)

  Implementation:

  1. Used `jest.fn()` to mock global `fetch`
  2. Simulated both resolved and rejected API responses
  3. Verified output against mock quiz data

- *OpenAI API (Bud-E) –* **`/utils/openaiApi.js`**

Purpose: Generate explanations and chatbot replies based on quiz data.

Tested:

1. POST request format, including proper prompt generation
2. Handling of both valid and malformed server responses
3. Graceful fallback when the explanation is missing

Implementation:

1. Mocked fetch calls with controlled return values
2. Asserted correct prompt construction using `questions[index]`
3. Ensured consistent return of `data.explanation` or `undefined`

These tests ensured that **data dependencies were decoupled from rendering logic**, making the app more resilient and reliable.

How It Was Implemented:

- All test files are organized under `src/tests`.

- Each test file:
    1. Wraps components in `MemoryRouter` and a mocked `QuizContext.Provider`
    2. Sets up mocks (localStorage, fetch, timers, DOM methods) before each test
    3. Cleans up state using `beforeEach` and `afterEach`

- We used:

    1. `jest.useFakeTimers()` for animations and timeouts
    2. `jest.mock()` and `jest.fn()` for API mocking
    3. Assertions on navigation and loading states

Results:

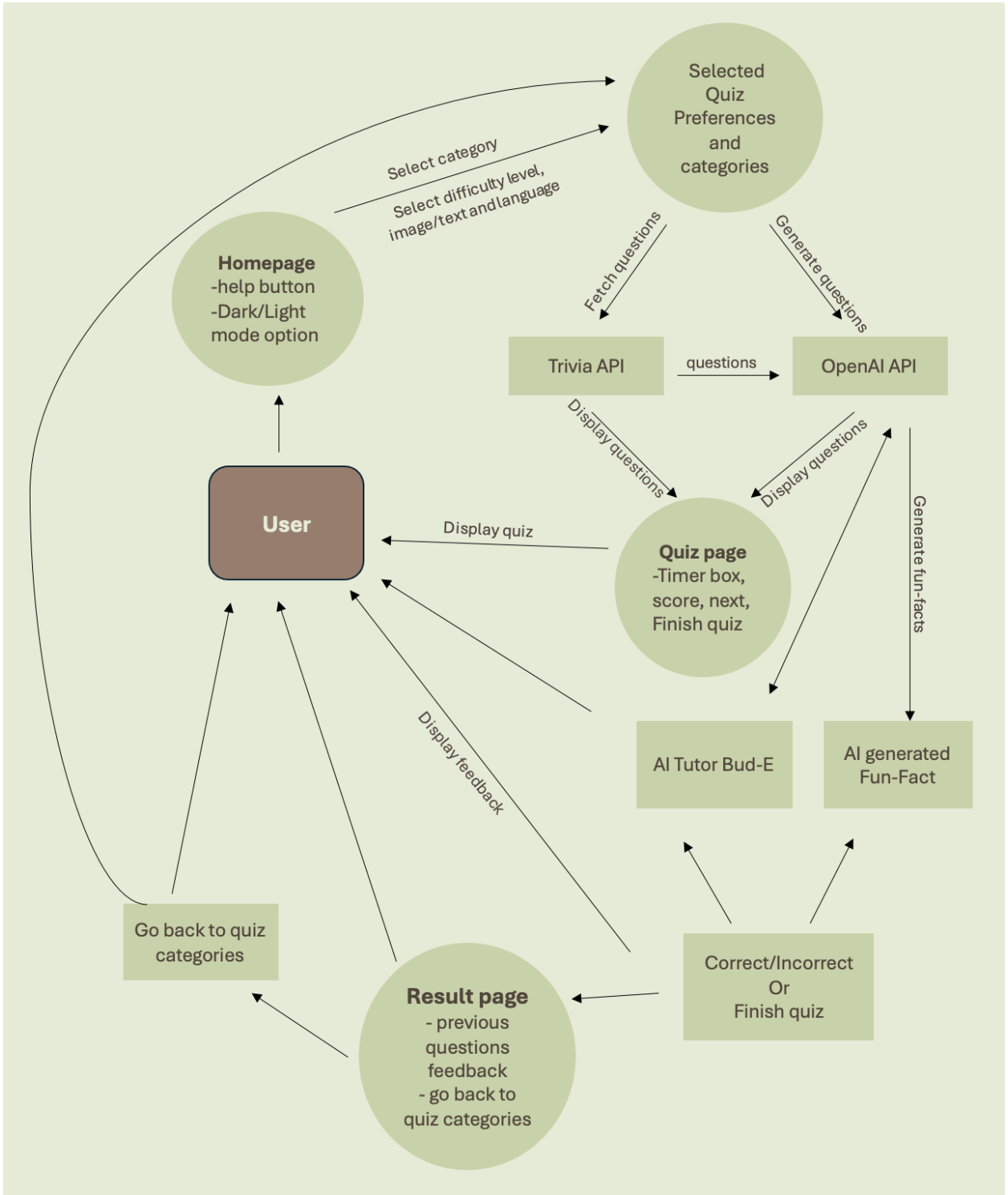
- High test coverage across key components and flows.
- Found and fixed early bugs in navigation, animation, and context sync.
- Built a strong testing foundation that is scalable and easy to extend for future features.
- Ensured stable behavior under edge cases like:

    - Empty quiz question lists
    - Missing explanation responses
    - Network/API failures

- Built a strong testing foundation that is scalable and easy to extend for future features.

Project Architecture Diagrams

Updated Level 1 DFD- BrainGoated:



The Level 1 DFD provides a comprehensive view of the data flow within the BrainGoated system, representing how user inputs, APIs, and internal pages interact to deliver quiz content, AI-powered enhancements, and feedback. This updated DFD reflects all newly implemented features, including multilingual quizzes, AI-generated fun facts, chatbot explanations, and responsive UI controls like dark/light mode toggling.

External Entity: User

The User is at the center of interaction. They can:

- Select quiz preferences like category, difficulty, type (text/image), and language.
- Navigate between homepage, quiz page, and result page.
- View feedback and interact with Bud-E chatbot.
- Switch between light/dark mode as needed.

Homepage:

- Provides a starting point for the user.
- Includes a help button for guidance and a dark/light mode toggle for better accessibility and UI customization.
- Sends the user to the preference selection process.

Selected Quiz Preferences and Categories:

- The user customizes their quiz by selecting:
    - Category (e.g., Science, History, Custom Quiz etc)
    - Difficulty level (Easy, Medium, Hard)
    - Question format (Text/Image)
    - Language (Supports multilingual quizzes)
- These preferences are passed to both the Trivia API and OpenAI API depending on the quiz type.

Trivia API:

- Responds to the selected preferences by returning quiz questions.
- Supports both text-based and image-based questions.
- Questions are passed to the Quiz Page for display.

OpenAI API:

- Used when:
    - Generating custom quizzes based on niche topics.
    - Creating AI-generated fun facts after each question.
    - Providing follow-up explanations through the chatbot.
- These responses are passed into the Quiz Page and Bud-E chatbot system.

Quiz Page:

- Central hub where users take the quiz.
- Includes:
    - Timer box
    - Score display
    - Navigation (Next / Finish quiz)
- Displays questions from either the Trivia API or OpenAI API.
- Sends user answers to trigger AI response generation and fun fact display.

AI Generated Fun-Fact:

- After a user answers a question, the system sends a query to OpenAI for a fun fact or extra context.
- This response is presented in the Correct/Incorrect feedback popup.

AI Tutor Bud-E:

- Allows users to ask follow-up questions about quiz topics.
- Sends prompts to the OpenAI API and displays conversational responses.
- Provides a more personalized learning experience.

Correct/Incorrect or Finish Quiz:

- When a user submits an answer, this process evaluates the result.
- Displays the correct answer, user choice, and a fun fact or explanation.
- If all questions are answered, transition to the Result Page.

Result Page:

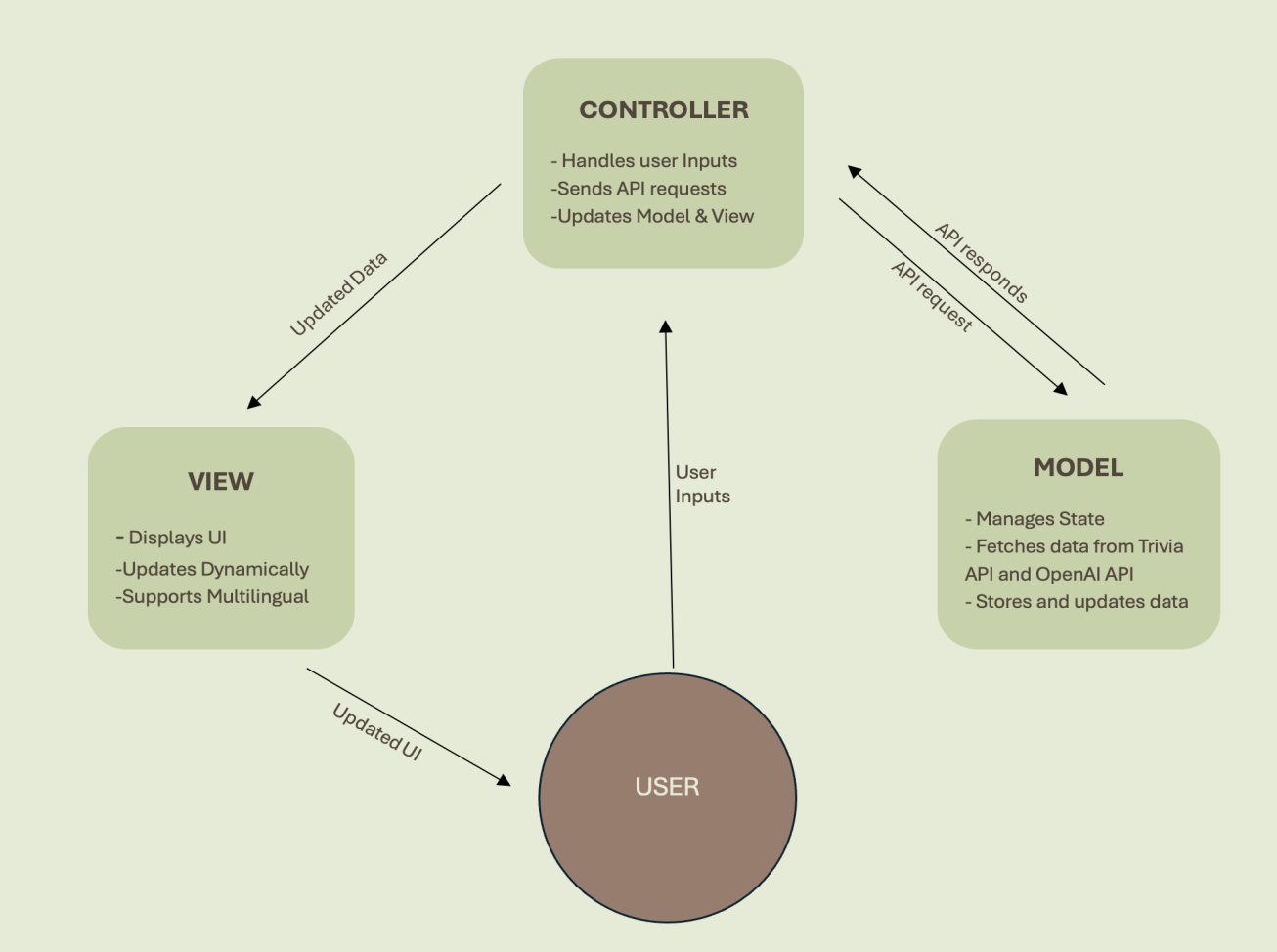
- Displays:
    - Feedback on previous questions
    - Final score

    - Option to go back to quiz categories to restart
- Provides closure to the quiz session and a way to loop back to the quiz categories.

Go Back to Quiz Categories:

- Allows users to restart a quiz session with new settings.
- Brings users back to the preference selection process.

Updated Model View Controller (MVC) Model:



Component Breakdown:

- *Model (M):*
    - Manages frontend application state (questions, answers, score, timer, chatbot history).
    - Fetches:
        i. Quiz questions from *The Trivia API* (text/image, in selected language).
        ii. AI-generated fun facts, explanations, and custom quizzes from *OpenAI API* (in selected language) and chatbox for interaction from *OpenAI API*.
    - Stores and updates all dynamic data including multilingual settings.
- *View (V):*
    - UI Components built in React.
    - Displays quizzes, score, feedback, and chatbot conversations.
    - Supports multiple languages dynamically via props/state updates.
    - Responsive and styled with CSS animations and motion effects.
- *Controller (C):*
    - Handles user actions (selecting category, difficulty, language, quiz type).
    - Sends requests to APIs based on user input.
    - Updates the model and triggers UI updates.
    - Manages logic for fun fact generation, language preferences, and fallback handling.
- *User:*
    - Selects quiz parameters (category, difficulty, language, type).
    - Takes quizzes, receives feedback, interacts with Bud-E chatbot.
    - Optionally creates custom quizzes in selected languages.

Known Bugs and Issues

Description	Steps to Produce	Severity	Github Issue
Limited number of questions for Image Quizzes in the Science Category	Go to Homepage> Scroll down to Quiz Categories > Select image quizzes> Select Science category > Start quiz. Only 3 questions are available due to limited API data.	Medium/High	issue #107
Leaf animation around Bud-E appears cut off at the edges	Observe Bud-E's idle animation on the homepage; note that the animated leaf sometimes exceeds its container, causing it to appear clipped	Low	issue #108
Repetition of questions in Image Quizzes	Select any category with Image Quizzes > Start quiz multiple times. Some questions repeat frequently due to limited pool of image-based questions from the API.	Medium	issue #109
Irrelevant images shown in some categories (e.g., History)	Go to Homepage > Scroll down to Quiz Categories > Choose any difficulty and language > Choose Image Quiz > Choose History Category > Start quiz. Some questions show modern or irrelevant images (e.g., unrelated pop culture content or images belonging to current time).	Low	issue #111
Lack of variety in image quiz content	Go to Homepage > Scroll down to Quiz Categories > Choose any difficulty and language > Choose Image Quiz > Choose General Category > Start quiz. Most images are about dogs.	Medium	issue #112

Sometimes images in the image quiz questions take long to load	Go to Homepage > Scroll down to Quiz Categories > Choose any difficulty and language > Select any category with "Image Quiz" enabled. Observe a delay in image loading or question rendering.	Medium	issue #113
Bud-E occasionally returns inappropriate answers	Start a Quiz with any category, language and difficulty selections > Answer a quiz question > Ask Bud-E a sensitive or tricky question in the chat. Despite kid-friendly prompting, inappropriate responses can occur.	High	issue #114
Difficulty levels from the Trivia API are not consistently accurate	Go to Homepage > Scroll down to Quiz Categories > Select "Easy" difficulty from the quiz settings. Some questions returned appear too challenging, even for adults.	Medium/High	issue #115
Image questions do not vary with difficulty level	Go to Homepage > Scroll down to Quiz Categories > Select different difficulty levels (e.g., "Easy","Medium", and "Hard") for an image quiz. The same or very similar questions are often returned regardless of the selected level.	Medium/High	issue #116
Difficulty setting often has little effect on text question difficulty	Go to Homepage > Scroll down to Quiz Categories > Choose different difficulty levels for a text-based quiz > Select any Category and Start Quiz > Observe that the questions appear to be of similar difficulty across all levels, due to the limited pool of image-based questions from the API.	Medium/High	issue #117
Sometimes inappropriate song-related questions appear in the Music category	Go to Homepage > Scroll down to Quiz Categories > Choose any difficulty and language > Select “Text” mode > choose the Music category, and occasionally observe questions referencing mature or explicit song lyrics	High	issue #118

Future Work and Potential Improvements

While all of the M1 goals were achieved, a few features are still open for expansion.

1. ***User Progress Tracking / History:***
   Save user quiz history or performance to track improvement. Use localStorage or a backend (e.g., Firebase) to store scores across sessions.

2. ***Multiplayer or Timed Challenges:***
   Create socket-based multiplayer or leaderboard via backend integration.

3. ***Custom Quiz Builder (AI-powered):***
   Bring back the teacher-style feature by letting users build/share custom quizzes. Use OpenAI to structure quiz templates, then store/share via a database.

4. ***Accessibility Improvements:***
   Add ARIA roles, better color contrast, keyboard navigation support for a more inclusive UX.

Lessons Learned and Project Takeaways

Key Learnings:

- **React State Mastery**
  Our team deepened our understanding of useState, useEffect, and component-based architecture in React.
- **API Integration and Management**
  Learning how to structure API requests, handle async behavior, and implement fallbacks for errors was crucial.
- **Prompt Engineering**
  Optimizing prompts for OpenAI to produce relevant, concise, and appropriate responses was surprisingly challenging and rewarding.
- **Responsive**
  Frontend strategies for responsive UI design and state-driven UI logic.

- **Agile-Kanban Execution**
  Regular updates via GitHub issues, clear milestones, and task assignments helped avoid last-minute crunches.

Challenges Faced and How We Overcame Them:

Challenge	Solution
Trivia API didn't support custom quiz creation	Removed the teacher page and instead used OpenAI to generate customized quizzes dynamically.
Long delays from OpenAI API responses	Added loading states and optimized prompt sizes to reduce cost and latency.
AI responses were occasionally irrelevant or repetitive	Iteratively refined prompt structure and used response truncation to keep them concise.
Complex conditional UI logic (pop ups, feedback, timer)	Broke UI into smaller, testable components and used useContext for shared state.
Mobile layout issues with Bud-E and popups	Used media queries and conditional CSS classes to improve responsiveness.
API Rate Limits	Used delay functions and batching strategies where possible.

Appendix

A. Group Members and Contributions:

This section outlines the contributions of each group member to the Project and the Final Report.

Manjari:

Project Contributions

- Trivia API Integration:
  Integrated The Trivia API into the app, allowing users to select quiz questions by category, difficulty, and supported languages.

- Custom AI Quiz (OpenAI Integration):
  Developed the “Build Your Quiz” feature using OpenAI, enabling users to generate personalized quizzes based on selected difficulty, category, and language.

- Fun Fact Generation:
  Implemented an AI-powered feature that generates a fun fact related to each custom quiz topic, enhancing user engagement and educational value.

- Responsive Design:
  Optimized the layout and user interface to be fully responsive across devices, including phones, tablets (iPad), and iMacs, ensuring a consistent experience.

- UI Enhancements:
  Improved the visual appeal and usability of the app through layout refinements, styling consistency, and interactive animations.

- Footer & Help Button:
  Created a fixed footer with important navigation links and implemented a floating help button to improve user guidance and accessibility.

- Final Presentation:
  Designed and edited the final presentation slides, including the addition of subtitles for improved accessibility and clarity.

Final Report Contributions:

- User Needs Analysis:
  Wrote the "User Needs and Fulfilment" section, outlining how BrainGoated meets the needs of students, parents, and teachers through its key features.

- Testing & Feedback Evaluation:
  Authored the "User Feedback and Testing Session" section, analyzing peer feedback from classmates and insights from real children in the app's target demographic.

- SDLC Documentation:
  Wrote the "SDLC Model" section, documenting the use of Agile–Kanban methodology and detailing how the team tracked progress, adapted plans, and integrated feedback.

- API Features & Updates:
  Contributed to the "API Features and Changes" section by explaining the integration and evolution of The Trivia API and OpenAI API from Milestone 1 to 2.

Beyzanur:

Project Contributions

- Prototyping:
  Designed the mid-fidelity prototype to guide initial development and layout planning.
- State Management & App Logic:
  Implemented the core quiz logic and state management system using React Context, ensuring smooth handling of question flow, user answers, score tracking, and category/difficulty settings.
- OpenAI Integration:
  Integrated the OpenAI API to power the Bud-E chatbot, enabling dynamic

follow-up questions and AI-generated educational content.

- Trivia API & Image Quiz Support:
  Connected and configured the Trivia API to fetch quiz data, including implementing the image-based quiz feature for enhanced interactivity.

- Bud-E & UI Animations:
  Designed and implemented animations for the Bud-E character and created the growing flower animation for the loading screen to enhance user experience.

- CI/CD Pipeline & Deployment:
  Set up the CI/CD pipeline using GitHub Actions and Netlify CLI for automatic testing and deployment. Integrated UptimeRobot for continuous site monitoring.

- UI Styling Support:
  Assisted in refining component styling and ensuring responsive design compatibility across devices whenever needed.

Final Report Contributions

- CI/CD Pipeline Explanation:
  Documented and illustrated the CI/CD workflow in the final report, including automated testing, build processes, and deployment.

- Bug Tracking:
  Created the bug report table and uploaded issues to GitHub with clear reproduction steps, context, and severity labels.

Alisha:

Project Contributions

- Website Structure & Design:
  Built the base structure of the BrainGoated website and designed the core UI layout.

- Styling & Theming:
  Applied consistent styling across components and implemented dark mode support.
- Responsive Design:
  Ensured the website looks great on mobile, tablet (iPad), and desktop using media queries and flexible layouts.
- Testing:
  Thoroughly tested component rendering, navigation flow, and UI responsiveness.
- UI Enhancements:
  Polished the user experience with layout tweaks and improved component styling.
- Demo video:
  Created the demo video showcasing BrainGoated's features and user experience.

Final Report Contributions

- Authored the “Key Learnings” section, summarizing the team’s major technical and collaborative takeaways.
- Wrote the “Challenges Faced & Solutions” part, documenting real-world obstacles and how the team overcame them.
- Contributed to the “Future Improvements” section by identifying opportunities for scalability and future growth.
- Designed the MVC (Model View Controller) and explained its components and application flow in the report.

Gurpreet:

Project Contributions

- Website Structure & Design:
  Designed and implemented the base structure of the website.
- Styling & Theming:
  Applied consistent styling across components
- UI Enhancements:
  Contributed to styling and UI enhancements, including Dark Mode styling and responsiveness.
- Testing:
  Tested each page thoroughly, verifying component rendering, interactions, and state behavior.

Tested all API integrations, ensuring correct data flow and error handling for Trivia and OpenAI APIs.

- Demo Video:
  Created the demo video showcasing BrainGoated's features and user experience.

Final Report Contributions

- Authored the detailed section on our Testing Strategy, covering tools, philosophy, and implementation.
- Designed the DFD (Data Flow Diagram) and explained its components and application flow in the report.

B. Changelog – Revisions Since M1:

Milestone 1 Revision	The “quiz creation” feature originally planned for teachers was removed due to Trivia API limitations.
Milestone 1 Revision	The "Learn First" feature was consolidated into dynamic fun facts to avoid disrupting the quiz flow.
Milestone 2 Revision	Language selection was introduced based on accessibility goals for both normal quizzes and AI-generated quizzes.